3. Linked Lists

- Motivation
- Pointers in C++
- Dynamic Size Arrays
- Linked List Nodes
- Connecting Nodes
- Linked List Operations
- Print List
- Create List
- Insert List
- Multiple Insertions
- Insert at End
- Sorted Insert
- List Empty?
- List Full?
- Searching A List
- Delete From List
- Delete Code
- Destroy List
- Implementation Examples
- Using Head and Tail Pointers
- Using Doubly Linked Lists
- Insertion into Doubly Linked List
- Deletion from Doubly Linked List

Motivation

- Arrays fast to access but fixed size.
- Linked list invented for dynamic data of variable length.
- Code is more complex and often slower.
- Programmers must make space/time trade-off when selecting data structure.

Pointers in C++

float x;	variable contains float value
float *y;	variable contains address of a float variable
y = &x;	initializes y to have address of x
*y = 4.2	stores 4.2 in variable x
y = new float;	allocates space for new variable on heap
delete y	gives space back to system
*y = 1.7;	illegal after delete called, can make program crash

Dynamic Size Arrays

int *data;	integer pointer
int size = 42	variable contains address of a float variable
data = new int[size]	allocates array of integers on heap
data[7] = 19	use array normal way
*(data + 7) = 91	pointer arithmetic used to access array. (faster but ugly)
delete []data;	gives space back to system (failure to delete causes memory leak)

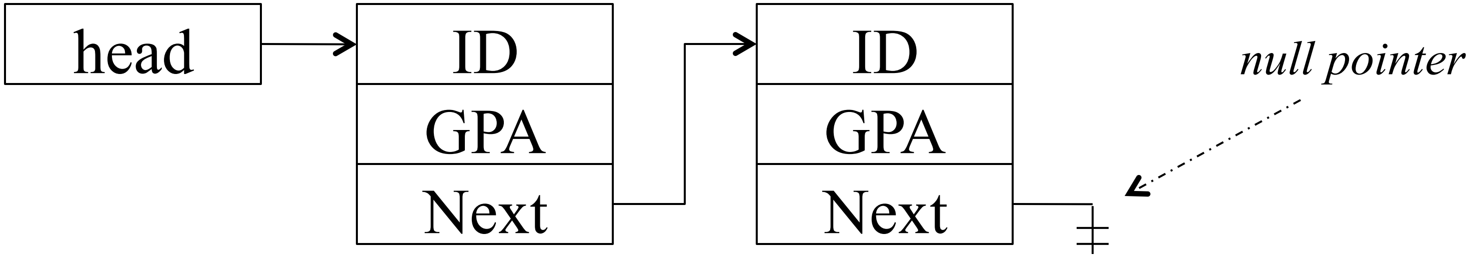
Linked List Nodes

- We create dynamic size data structure by chaining together chunks of memory.
- A linked list node contains N data elements and 1 or 2 pointers to other data in the list

```
struct Node
{
  int ID;
  float GPA;
  Node * Next;
}
```

- ➢ data fields
- ➢ pointer to another Node record

Linked List Nodes



- We draw nodes as boxes and connect nodes with arrows from pointer fields.

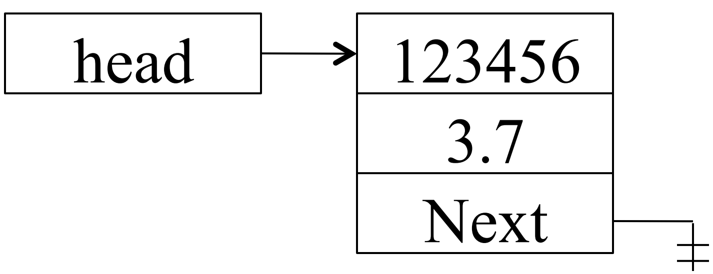
Connecting Nodes

Node * ptr
ptr = new Node;

- pointer to Node record
- allocate space on heap

ptr->ID = 123456;
ptr->GPA = 3.7;
ptr->Next = NULL;

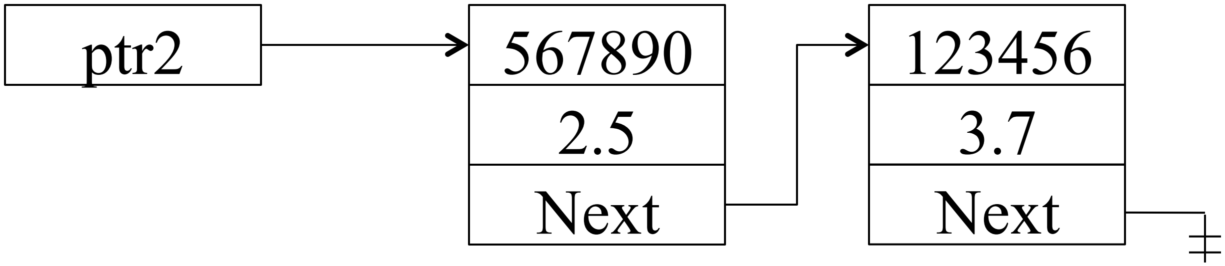
➢ set data fields



Connecting Nodes

```
Node * ptr2
ptr2 = new Node;

ptr2->ID = 567890;
ptr2->GPA = 2.5;
ptr2->Next = ptr;
```

- create another node and add to chain

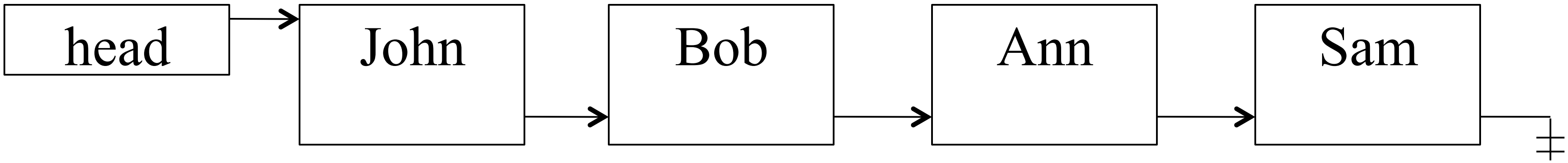


Linked List Operation

- Linked list do not allow random access like arrays so we must write functions to store and retrieve data.
- A typical linked list ADT has the following operations:
  - print list
  - create list
  - insert data into list
    - sorted order
    - unsorted order
  - list empty?
  - list full?
  - search list for data
  - delete data from list
  - destroy whole list

Print List

- Assume we are given linked list of names.



- The variable head gives us address of first node, but we don't have direct pointers to other nodes in list.

```
Node * ptr = head;
while (ptr != NULL)
{
   cout << "name:" << ptr->name << endl;
   ptr = ptr->next;
}
```

- Code above uses a temporary pointer ptr to "walk the list" and access each node to print the data.

Create List

- Creating an empty list is trivial, we just declare a head pointer and set it to null.

- Creating a copy of another list or the data in an array or data file requires us to <u>insert</u> data into a list

```
Node *head;          // define head pointer

head = NULL;         // set to null value

Node *head = NULL; // preferred method
```

Insert List

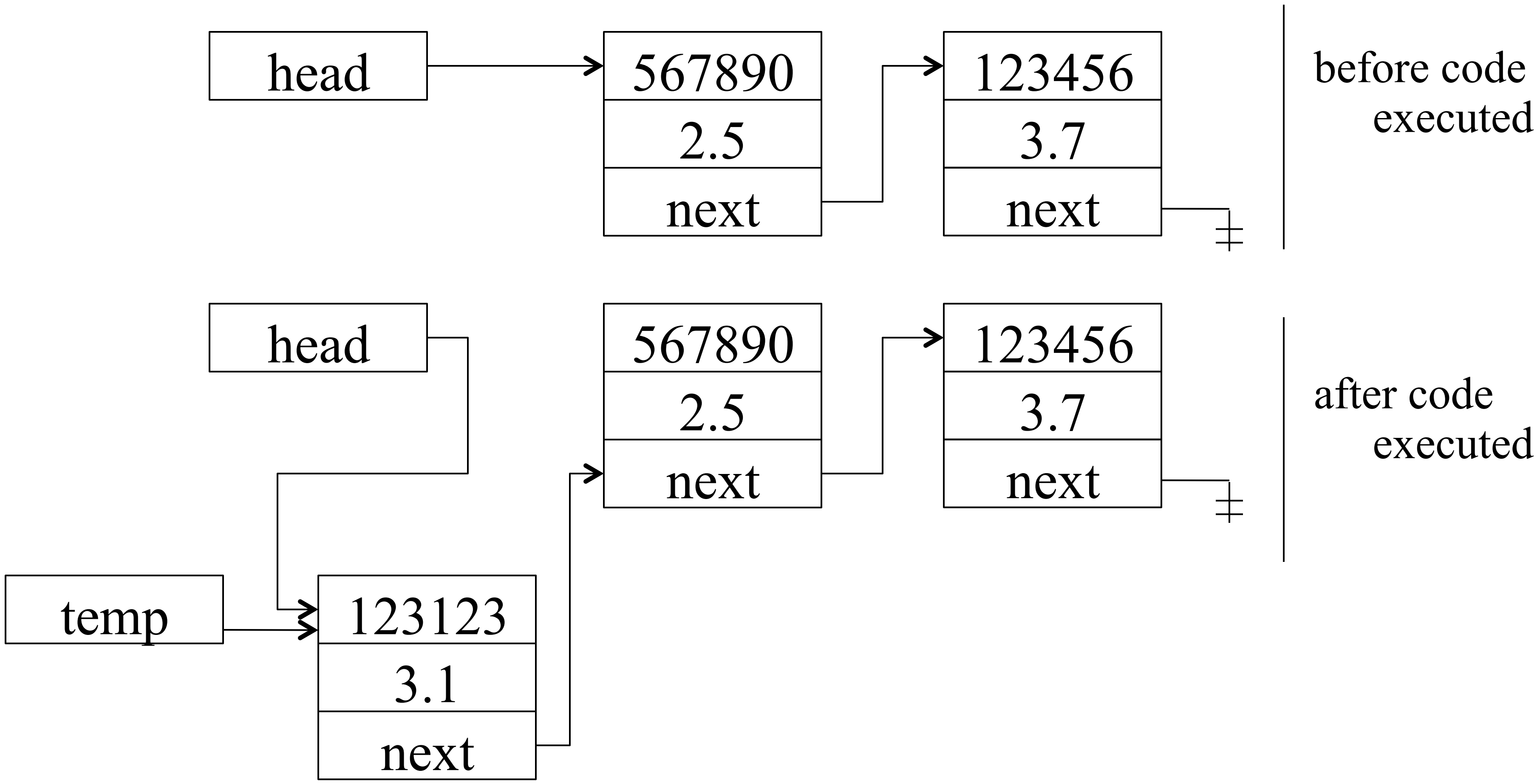
- Easiest place to connect a new node is <u>before</u> the head of the current linked list

```
Node *temp;
temp = new Node;
temp->ID = 123123;
temp->GPA = 3.1;
temp->next = head;
head = temp;
```

➢ create new node

➢ linked before head of list

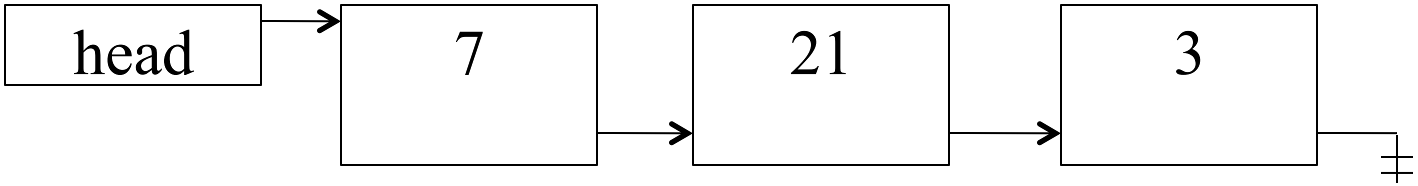
Insert List


head
567890
2.5
next
123456
3.7
next
before code
executed
head
567890
2.5
next
123456
3.7
next
after code
executed
temp
123123
3.1
next

Multiple Insertions

- Assume we are given linked list of names.

```
insert(3);
insert(21);
insert(7);
```



- Can also insert data in specified locations (eg. after node N) or in sorted order based on the data.
- Requires much more care to get links between nodes correct

Insert at End

- Can insert node <u>after</u> last node in list.
- We must traverse the list to get to the last node.

```
Node * temp = head;
while ((temp != NULL)&&(temp->next != NULL))
   temp = temp->next;
if (temp != NULL)
{
   temp->next = new Node;
   temp->next->num = 42;
}
else
{
   head = new Node;
   head->num = 42;
}
```

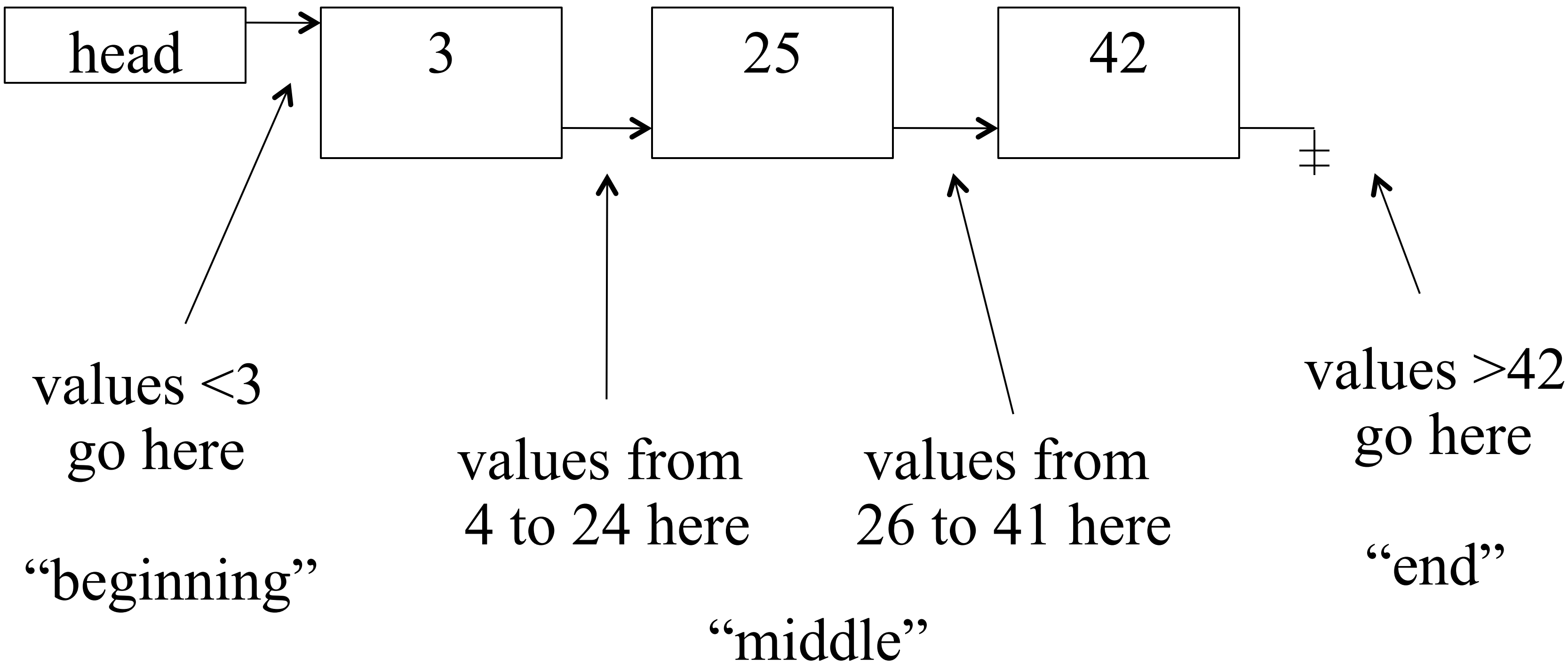
normal case.

special case for empty list

- Code above stops when temp is pointing to the last node in the linked list.

Sorted Insert

- To keep data in sorted order we can search list for insertion point prior to insertion.



Sorted Insert

- Our code must handle all <u>cases</u> above.

```
Node *temp = head;
while ((temp != NULL))&&(temp->num < value))
  temp = temp->next;
```

- Code above sets temp to node <u>after</u> our insertion point.
- No way to go back (yet) so we need to keep track of more information.

Sorted Insert

```
Node *temp = head;
node *prev = temp;
while ((temp != NULL))&&(temp->num < value))
{
   prev = temp;
   temp = temp->next;
}
if ( insert at head )
{
   head = new Node;
   head->num = value;
   head->next = temp;
}
else
{
    prev->next = new Node;
   prev->next->num = value;
   prev->next->next = temp;
}
```

- What condition do we need above?

List Empty?

- Trivial to check if list is empty since we set head to NULL before data is inserted

```
bool list_empty()
{
   return (head == NULL);
}
```

List Full?

- We don’t need to worry about array bounds when adding data to a linked list (the whole point of this data structure)
- Systems can run out of heap space, so eventually a list can become full.
- We need to check if new returns NULL everywhere in code to detect this.

Searching a List

- Can easily traverse a list to see if desired data is there.

```
Node *temp = head;
while ((temp != NULL))&&(temp->num != value))
   temp = temp->next;
```

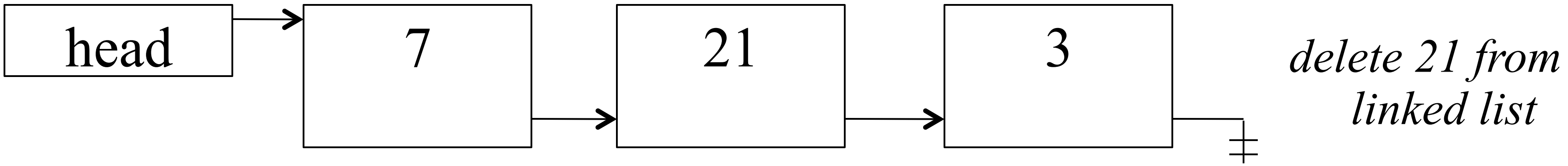
- This loop stops when data is found or when temp has “fallen of the list”.

```
if (temp == NULL)
   found = false;
else
   found = true;
```

- Look up time for linked list is about the same as scanning an unsorted array for a data value.
- There is no way to do a binary search on a sorted linked is (the main reason binary search trees were invented years ago).

Delete from List

- Often need to remove data from a dynamic data structure



- We can do this by adjusting the pointers (and calling delete to return space to heap)/
- Tricky parts:
    - finding node to delete
    - change pointers in correct order
    - handle special cases

Delete Code

```
Node *temp = head;
node *prev = NULL;
while ((temp != NULL)&&(temp->num != value))
{
   prev = temp;
   temp = temp->next;
}
```

- Now prev should point to node before node to be deleted, temp should point to node to be deleted
- If temp is null, the data was not found and can't be delete.

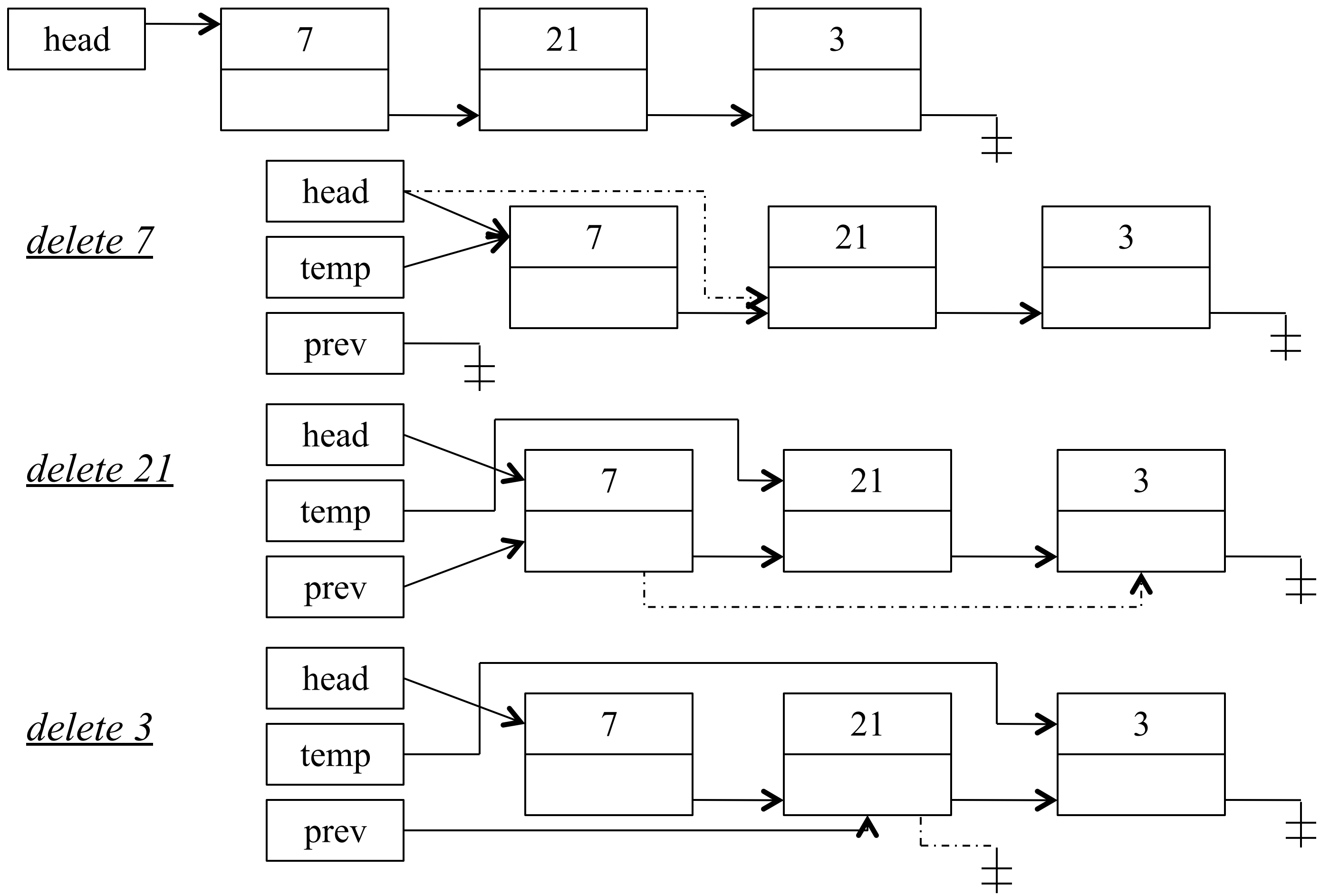
```
if ((temp != NULL)&&(temp == head))
{
   head = temp->next;
   delete temp;

}
else if (temp != NULL)
{
   prev->next = temp->next;
   delete temp;
}
```

beginning

middle or end

Delete Code

- Should verify that code can delete node at front, middle, and end of list.



Destroy List

- When we are finished with a linked list it is essential to give the space back to the system.
- Otherwise your program will leak memory and eventually the program will crash (or the system will).

```
Node *ptr = head;
while (ptr != NULL)
{
   head = head->next;
   delete ptr;
   ptr = head;
}
```

switching order here could cause bugs!

Implementation Examples

<u>Using Head Pointer</u>

- All examples so far have used a head pointed to first node as the <u>only</u> way to access a list.
- Any data can be stored in Node by changing struct. (See examples).
- Can also use class to store data in a Node object.

```
class Node
{
public:
   int ID;
   float GPA;
   Node *next;
}
```

- same effect as using a struct.
- No information hiding or data checking.

Implementation Examples

Using Node Objects

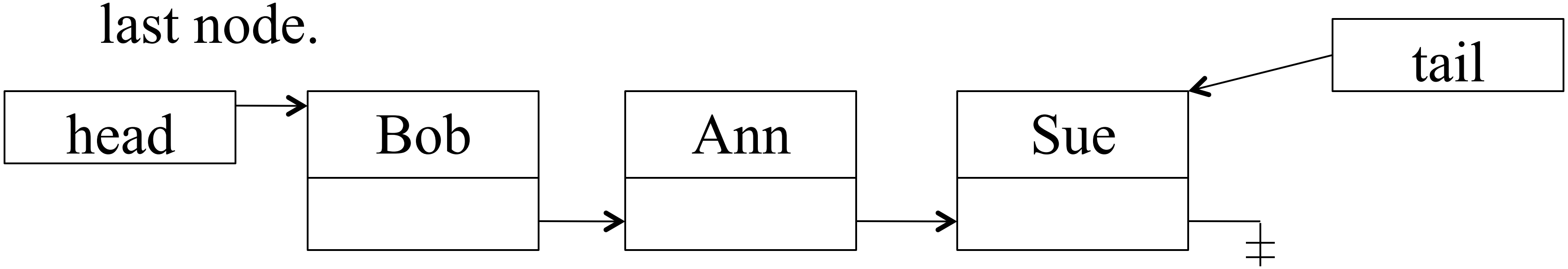
- With true OOP approach the data should be private:

```
class Node
{
public:
   Node();
   ~Node();
   int getID();
   int getGPA();
   Node *getNext();
   void setID(int id);
   void setGPAT(float gpa);
   void setNext(Node *next);
private:
   int ID;
   float GPA;
   Node *next;
};
```

- This way the setID can check number between 000000 and 999999 and GPA is between 0.0 and 4.0.

Using Head and Tail Pointers

- If we add to end of list a lot, it is handy to have pointer to the last node.



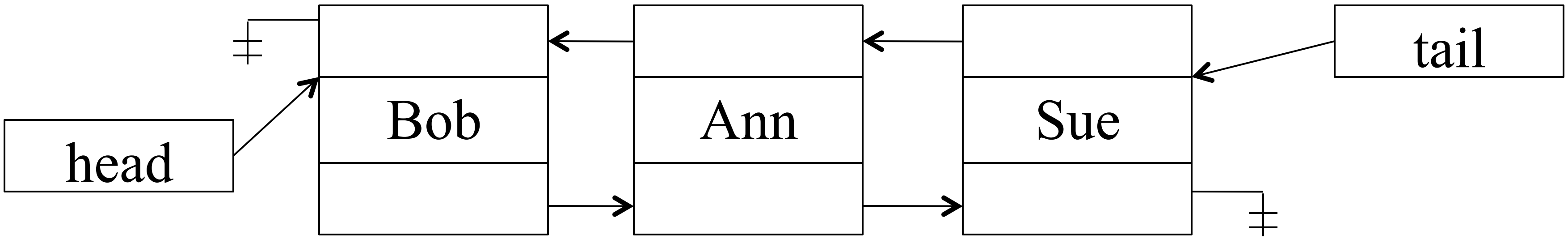
- Now insert at end is trivial. (almost)

```
tail.next = new Node;
tail = tail.next;
tail.name = "John";
tail.next = NULL:
```

- Need to handle special case of insert into an empty list.
- Need to modify delete code too whenever the tail node is deleted we move pointer to the previous node (or set to NULL when the last node is deleted).

Using Doubly Linked Lists

- In order to walk a list in either direction, we need two pointers per Node, and a tail pointer.

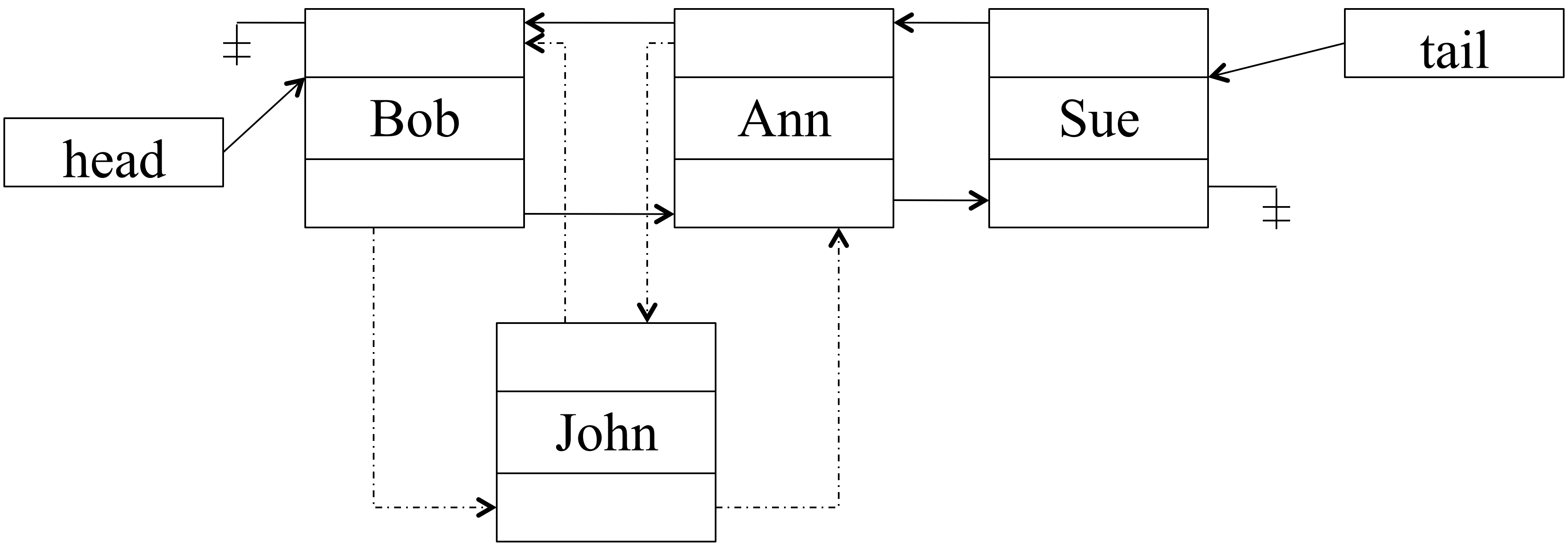



```
struct Node
{
   string Name;
   Node *next;
   Node *prev;
};
```

These are not reserved words but very common

Insertion into Doubly Linked Lists

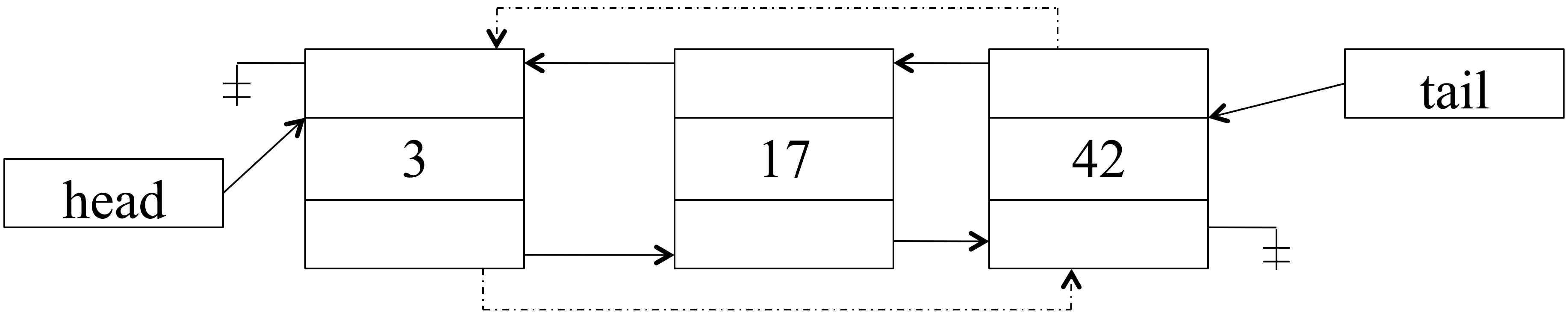
- Must consider insert at head, tail, and in middle.



- Must now update 4 pointers (dotted lines above) in the correct order to link in the node into the list.

Deletion from Doubly Linked Lists

- Again head, tail, and middle must be handled



- Only need to update 2 pointers to jump over deleted node.

```
Node *ptr = head;
while((ptr != NULL)&&(ptr->num !=17))
   ptr = ptr->next;
if ((ptr != NULL)&&(ptr!=head)&&(ptr!=tail))
{
   ptr->prev->next = ptr-next;
   ptr->next->prev = ptr->prev;
}
```